Directory Size Calculator – Project Report

Author: *Sarthak Jain*
Date: *June 30, 2025*
GitHub Repository: https://github.com/sarthkjn/directory-size-calculator

1. Introduction

This project is a command-line C++ application that simulates a virtual file system and allows users to navigate folders, list contents, and calculate the total size of directories using recursion.

The key goal is to replicate basic file system functionalities (cd, ls, and size) without interacting with the actual file system. The project is designed to run fully in memory, making it ideal for learning recursive data structures, class design, and command-line application development.

2. Project Objectives

- Build an interactive in-memory directory structure.
- Support basic file system commands (cd, ls, size).
- Use recursive logic to calculate the total size of a directory.
- Implement clean object-oriented C++ code.
- Create a modular structure with test coverage and CMake support.
- Include unit tests and documentation.

3. Features Overview

- **Virtual File System**: All data resides in memory (not on disk).
- **Recursion**: Calculates the total size of a folder including all nested files.
- **Navigation Support**: Users can move into (cd <folder>) and out of (cd ..) directories.
- **Directory Listing**: ls shows both files and subfolders.
- **Testable Code**: Unit tests validate behavior and correctness.
- **CMake Build Support**: Easier compilation and platform independence.

4. Project Folder Structure

directory-size-calculator/

```
├── include/            # Header files
│   ├── File.h
│   ├── Directory.h
│   └── FileSystem.h
├── src/                # Source files
│   ├── File.cpp
│   ├── Directory.cpp
│   ├── FileSystem.cpp
│   └── main.cpp
├── test/               # Unit tests
│   └── test_filesystem.cpp
└── README.md   # Project documentation
└── CMakeLists.txt # CMake build config
├── /assets/            # Screenshots
│   ├─── cli-demo.png
│   └── test-demo.png
```

Each directory and file is carefully structured to support clean modular development.

5. Key Classes and Responsibilities

5.1 File Class

Purpose: Represents a file with a name and size.

Header Declaration (File.h):

```
class File {
private:
    std::string name;
    int size;

public:
    File(const std::string& name, int size);
    std::string getName() const;
    int getSize() const;
};
```

Explanation:

- The constructor initializes a file with a name and size.
- getName() and getSize() provide read-only access.

5.2 Directory Class

Purpose: Represents a folder that may contain:

- Files (std::vector<File>)
- Subdirectories (std::map<std::string, std::unique_ptr<Directory>>)
- Pointer to its parent (for cd .. navigation)

Key Methods:

- addFile(): Adds a file to the directory.
- addSubdirectory(): Adds a new subdirectory.
- getSubdirectory(): Finds a subdirectory by name.
- calculateSize(): Recursively calculates total size.

Recursive Function Example:

```
int Directory::calculateSize() const {
    int totalSize = 0;

    for (const auto& file : files) {
        totalSize += file.getSize();
    }

    for (const auto& pair : subdirectories) {
        totalSize += pair.second->calculateSize();  // recursive call
    }

    return totalSize;
}
```

5.3 FileSystem Class

Purpose: Core interface between the user and the directory structure.

Responsibilities:

- Maintain the root and current directory.
- Implement command handlers like ls, cd, and size.
- Seed the file system with test folders/files.

Key Members:

Directory* root;

Directory* current;

Command Example – Change Directory:

```
void FileSystem::changeDirectory(const std::string& dirname) {
    if (dirname == "..") {
        if (current->getParent() != nullptr) {
            current = current->getParent();
        } else {
            std::cout << "Already at root directory.\n";
        }
    } else {
        Directory* subdir = current->getSubdirectory(dirname);
        if (subdir != nullptr) {
            current = subdir;
        } else {
            std::cout << "No such directory: " << dirname << "\n";
        }
    }
}
```

5.4 main.cpp

Purpose: The main CLI loop.

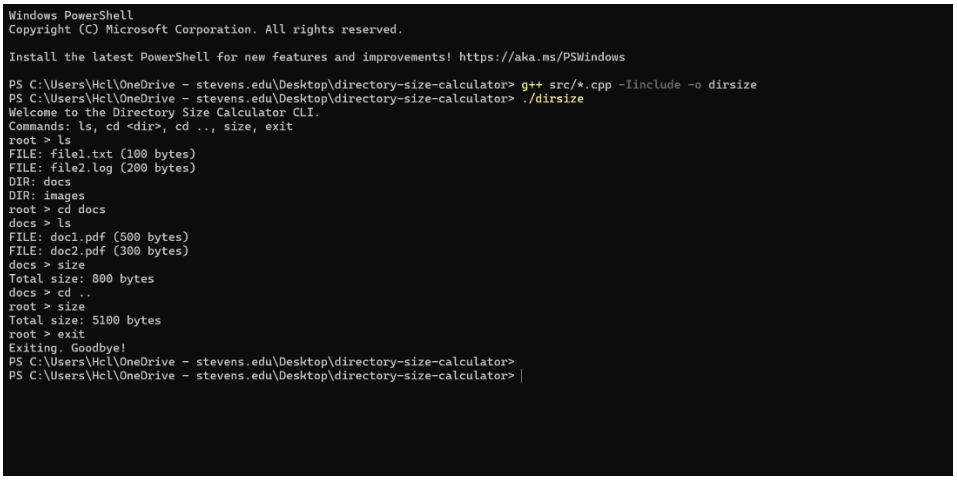
Functionality:

- Prints a prompt
- Reads user input
- Delegates command execution to FileSystem

```cpp
while (true) {
    fs.printCurrentPath();  // e.g., "root > "
    std::getline(std::cin, command);

    if (command == "ls") {
        fs.listDirectory();
    }
    else if (command == "size") {
        fs.showSize();
    }
    else if (command == "exit") {
        break;
    }
    else if (command.substr(0, 3) == "cd ") {
        std::string dirname = command.substr(3);
        fs.changeDirectory(dirname);
    }
    else {
        std::cout << "Unknown command. Try: ls, cd <dir>, cd .., size, exit\n";
    }
}
```

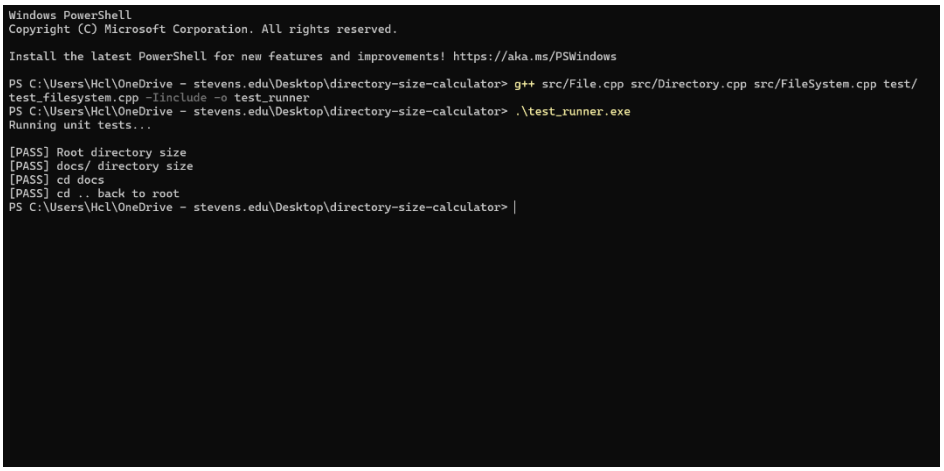
6. Sample CLI Execution



Description:

- Shows the use of ls, cd, and size
- Demonstrates directory traversal
- Displays total recursive size

7. Unit Testing



Tested Scenarios:

- Total size of the root
- Size of a specific subdirectory
- Navigation (cd, cd ..)

Tests are written in test_filesystem.cpp and built separately.

8. Building & Running the Application

Manual (g++):

g++ src/*.cpp -Iinclude -o dirsize

./dirsize

Unit Tests:

g++ src/*.cpp test/test_filesystem.cpp -Iinclude -o test_runner

./test_runner

Using CMake:

mkdir build && cd build

cmake ..

cmake --build .

9. Code-Level Walkthrough (All .cpp Files)

This section dives deep into the actual implementation files of the project. Each .cpp file is broken down by its major functions and logic. Comments, explanations, and usage examples are provided to clarify what each segment does.

9.1 File.cpp

Location:src/File.cpp
Associated Header: File.h

This file defines the methods for the File class — a simple structure representing a file's metadata.

```cpp
#include "File.h"
using namespace std;

File::File(const string& name, int size) : name(name), size(size) {}

string File::getName() const {
    return name;
}

int File::getSize() const {
    return size;
}
```

Constructor:

- Initializes a File object with a given name and size.
- Uses an initializer list to directly assign values.

Example:

File myFile("data.txt", 1200);

// Represents a file named data.txt that is 1200 bytes in size

Method: getName()

- Returns the name of the file.
- Declared const because it doesn't modify any member variables.

Method: getSize()

- Returns the size (in bytes) of the file.

9.2 Directory.cpp

Location:src/Directory.cpp
Associated Header: Directory.h

This file handles everything related to directory-level operations, such as holding subdirectories and files, and calculating size.

```
Directory::Directory(const std::string& name, Directory* parent)
    : name(name), parent(parent) {}

const std::string& Directory::getName() const {
    return name;
}

Directory* Directory::getParent() const {
    return parent;
}

void Directory::addFile(const File& file) {
    files.push_back(file);
}

void Directory::addSubdirectory(const std::string& dirname) {
    if (subdirectories.find(dirname) == subdirectories.end()) {
        subdirectories[dirname] = std::make_unique<Directory>(dirname, this);
    }
}

Directory* Directory::getSubdirectory(const std::string& dirname) {
    if (subdirectories.find(dirname) != subdirectories.end()) {
        return subdirectories[dirname].get();
    }
    return nullptr;
}

void Directory::listContents() const {
    for (const auto& file : files) {
        std::cout << "FILE: " << file.getName() << " (" << file.getSize() << " bytes)\n";
    }
    for (const auto& pair : subdirectories) {
        std::cout << "DIR: " << pair.first << "\n";
    }
}

int Directory::calculateSize() const {
    int totalSize = 0;

    for (const auto& file : files) {
        totalSize += file.getSize();
    }

    for (const auto& pair : subdirectories) {
        totalSize += pair.second->calculateSize();  // recursive call
    }

    return totalSize;
}
```

Constructor:

- Assigns the name and parent pointer.
- Enables recursive back-tracking with cd ...

Method: addFile()

- Adds a file object to the directory's files vector.

Example:

File file1("photo.jpg", 500);

dir->addFile(file1);

Method: addSubdirectory()

- Adds a new subdirectory using std::unique_ptr
- Automatically sets the current directory (this) as the parent

Method: getSubdirectory()

- Looks for a subdirectory by name.
- Returns pointer if found, nullptr if not.

Method: calculateSize()

- Adds the size of all files in the directory.
- Then recursively calls calculateSize() on each subdirectory.

Example:

root/

├── file1.txt (100)

└── docs/

  └── doc1.pdf (500)

root->calculateSize() = 100 + docs->calculateSize() = 100 + 500 = 600

9.3 FileSystem.cpp

Location:src/FileSystem.cpp
Associated Header: FileSystem.h

This file manages user interactions, command parsing, and virtual directory management.

Constructor:

- Initializes the root directory.
- Sets the current working directory to root.
- Calls seedSampleData() to add predefined files and folders.

```
FileSystem::FileSystem() {
    root = new Directory("root");
    current = root;
    seedSampleData();
}
```

Method: seedSampleData()

- Adds sample data to test the system.
- Populates root, docs, images, and raw folders.

```
void FileSystem::seedSampleData() {
    // Add files to root
    root->addFile(File("file1.txt", 100));
    root->addFile(File("file2.log", 200));

    // Add docs/
    root->addSubdirectory("docs");
    Directory* docs = root->getSubdirectory("docs");
    docs->addFile(File("doc1.pdf", 500));
    docs->addFile(File("doc2.pdf", 300));

    // Add images/ with raw/
    root->addSubdirectory("images");
    Directory* images = root->getSubdirectory("images");
    images->addFile(File("img1.png", 1500));

    images->addSubdirectory("raw");
    Directory* raw = images->getSubdirectory("raw");
    raw->addFile(File("raw1.cr2", 2500));
}
```

Method: listDirectory()

- Displays the contents of the current directory.
- Distinguishes between files and folders.

```
void FileSystem::listDirectory() const {
    current->listContents();
}
```

Method: changeDirectory()

- Changes the current working directory.
- Navigates back with '..', or forward to a valid subdirectory.

```
void FileSystem::changeDirectory(const std::string& dirname) {
    if (dirname == "..") {
        if (current->getParent() != nullptr) {
            current = current->getParent();
        } else {
            std::cout << "Already at root directory.\n";
        }
    } else {
        Directory* subdir = current->getSubdirectory(dirname);
        if (subdir != nullptr) {
            current = subdir;
        } else {
            std::cout << "No such directory: " << dirname << "\n";
        }
    }
}
```

9.4 main.cpp

Location:src/main.cpp
Purpose: Entry point and command processing loop

CLI Loop

```
while (true) {
    fs.printCurrentPath();  // e.g., "root > "
    std::getline(std::cin, command);

    if (command == "ls") {
        fs.listDirectory();
    }
    else if (command == "size") {
        fs.showSize();
    }
    else if (command == "exit") {
        break;
    }
    else if (command.substr(0, 3) == "cd ") {
        std::string dirname = command.substr(3);
        fs.changeDirectory(dirname);
    }
    else {
        std::cout << "Unknown command. Try: ls, cd <dir>, cd .., size, exit\n";
    }
}
```

- Runs continuously until exit is entered.
- Interprets user commands and delegates to FileSystem.

Example Interaction

root > ls

FILE: file1.txt (100 bytes)

DIR: docs

root > cd docs

docs > size

Total size: 800 bytes

10. Conclusion

The **Directory Size Calculator** project demonstrates how a hierarchical file system can be effectively simulated using modern C++. Through this command-line application, users can navigate virtual directories, list contents, and calculate folder sizes recursively — all while the data remains entirely in memory.

This project goes beyond writing functional code. It emphasizes clean architecture, modular design, and object-oriented principles:

- The File class models basic file attributes
- The Directory class handles recursive data structures and relationships
- The FileSystem class acts as a command router, enabling seamless user interaction

Moreover, the implementation of recursion in the calculateSize() method offers an elegant solution to traversing a nested structure — a core concept in both data structures and systems programming.

The codebase is supported by clear documentation, unit tests, and a build system using CMake, making it not only functional but also maintainable and extendable.

This report has aimed to break down the logic, structure, and flow of the program so thoroughly that even a reader unfamiliar with C++ should be able to follow the logic, appreciate the design, and understand the purpose of every major component.

This project embodies the principles of clarity, structure, and purpose — making it both an educational tool and a demonstration of real-world software design in action.